Final Report: Scrambled Soboĺ Sequences via Permutation

1 Introduction

The Soboĺ sequence [12, 13] is one of the standard quasirandom sequences, and is widely used in **Q**uasi-**M**onte **C**arlo (QMC) applications. QMC methods are a variant of ordinary **M**onte **C**arlo (MC) methods that employ highly uniform quasirandom numbers in place of the pseudorandom numbers used in ordinary **M**onte **C**arlo (MC) [3]. QMC methods are now widely used in scientific computation, especially in estimating integrals over multidimensional domains and in many different financial computations. The error of MC methods is asymptotically $\mathbf{O}(N^{-\frac{1}{2}})$, where N is the number of samples, while QMC methods can have an error bound which behaves as well as $\mathbf{O}((logN)^s N^{-1})$, for s-dimensional problems. While quasirandom numbers do improve the convergence of applications like numerical integration, it is by no means trivial to provide practical error estimates in QMC due to the fact that the only rigorous error bounds, provided via the Koksma-Hlawka inequality [3, 21, 25], are very hard to utilize. In fact, the common practice in MC of using a predetermined error criterion as a deterministic termination condition, is almost impossible to achieve in QMC without extra technology. In order to provide such dynamic error estimates for QMC methods, several researchers [21, 25] proposed the use of **R**andomized QMC (RQMC) methods, where randomness can be brought to bear on quasirandom sequences through scrambling and other related randomization techniques [4, 5, 9]. The core of RQMC is to find fast and effective algorithms to randomize (scramble) quasirandom sequences.

Besides providing practical error estimates, another byproduct of scrambled quasirandom numbers is that they furnish a natural way to generate quasirandom numbers in parallel and distributed applications. This is because an elegant solution to this problem is to take a single quasirandom sequence and to provide a differently scrambled version of this sequence to each process requiring quasirandom numbers. Moreover, QMC applications have high degrees of parallelism, can tolerate large latencies, and usually require considerable computational effort, making them extremely well suited to parallel, distributed, and even Grid-based computational environments. In these environments, a large QMC problem is broken up into many small subproblems. These subproblems are then scheduled on the parallel, distributed, or Grid-based environment. In a more traditional instantiation, these environments are usually a workstation cluster connected by a local-area network, where the computational workload is cleverly distributed. Recently, peer-to-peer or Grid computing, the cooperative use of geographically distributed resources unified to act as a single powerful computer, has been investigated as an appropriate computational environment for MC applications [15, 16]. The computational infrastructure developed for this was based on the existence of a high-quality tool for parallel pseudorandom number generation, the **S**calable **P**arallel **R**andom **N**umber **G**enerators (SPRNG) library [18, 17]. The extension of this technology to quasirandom numbers would be very useful.

Unlike pseudorandom numbers, there are only a few common choices for quasirandom number generation. However, by scrambling a quasirandom sequence, one can produce a family of related quasirandom sequences. Finding one or a group of optimal quasirandom sequences within this family is an interesting problem, as such optimal quasirandom sequences can be quite useful for enhancing the performance of ordinary QMC. The process of finding such optimal quasirandom sequences is called the derandomization of a randomized (scrambled) family of quasirandom sequences. In addition to providing more quasirandom sequences for QMC, derandomization can help us to improve the accuracy of error estimation provided by RQMC. This is due to the fact that one can find a set of optimal sequences within a family of scrambled sequence family, and use sequences within this set for error estimation.

The purpose of randomizing (scrambling) in QMC is threefold. Primarily, it provides a practical method to obtain error estimates for QMC based by treating each scrambled sequence as a different and

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2009		**00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Final Report: Scrambled Sobo l Sequences via Permutation	**W911NF-06-1-0514**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Florida State University,Tallahassee,FL,32306-4166	**; 47107-CS.2**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
U.S. Army Research Office, P.O. Box 12211, Research Triangle Park, NC, 27709-2211	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
	47107-CS.2

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**16**	

independent random sample from a family of randomly scrambled quasirandom numbers [21]. Thus, RQMC overcomes the main disadvantage of QMC while maintaining the favorable convergence rate of QMC. Secondarily, scrambling gives us a simple and unified way to generate quasirandom numbers for parallel, distributed, and Grid-based computational environments. Finally, RQMC provides many more choices of quality quasirandom sequences for QMC applications, and perhaps even optimal choices as a result of derandomization. Thus, a careful exploration of scrambling and derandomization methods coupled with library-level implementations will play a central role in the continued development and use of RQMC techniques.

In this paper, we propose a new Sobol' scrambling algorithm based on the permutation of several groups of bits from the individual Sobol' numbers. Most of the current scrambling methods either randomize a single digit at each iteration, or randomize a group of digits through linear operations. In contrast, our multiple-digit scrambling is efficient and fast because it permutes small groups of digits. We implemented this new Sobol' scrambling algorithm in the software context of the `SPRNG`library, because `SPRNG`not only generates parallel pseudorandom numbers, but it also provides an extensible object-orient interfaces for merging scrambled Sobol' sequences.

The remainder of this paper is organized as follows. In § 2, we give a brief introduction to quasirandom or low-discrepancy sequences and the Sobol' sequence in particular. Our scrambling algorithm is described in § 3. In § 3, we also discus the implementation of the algorithm and how it is implemented in the object-oriented environment provided by the `SPRNG`library. The results of our experiments designed to verify the performance of the algorithm are demonstrated in § 4. In § 5, we summarize our conclusions and describes our future work.

2 The Sobol' Sequence

In this section, we first discuss quasirandom sequences and the conventional measure used for evaluating the uniformity of low-discrepancy sequences. Then we define the Sobol' sequence and algorithm used to generate it are given. Then, a few general scrambling algorithms are presented.

2.1 Low-Discrepancy Sequences

Quasirandom numbers are produced by a deterministic sequence, and as we will see below, they are often used in quasi-Monte Carlo integration. The reason behind using deterministic sequences is that the independence of random numbers plays a secondary role to their uniformity in certain Monte Carlo calculations, so sequences with better uniformity properties may lead to smaller errors in certain applications. To quantify how uniform a given sequence is, we need a well-defined measure of uniformity. There are, in fact, many ways to define and measure uniformity, we here use the one based on Niederreiter's development of the topic [20].

Let I^s denote the s dimensional unit cube. An infinite sequence $\{x_n\}$ in I^s is called uniformly distributed if for all measurable subsets J of I^s

$$lim_{N\to\infty}\frac{1}{N}\sum_{n=1}^{N}\chi_J(x_n) = m(J), \tag{1}$$

holds, where χ_J is the characteristic function of J, and $m(J)$ is the volume of J. Thus in the limit of an infinite number of points, every region in I^s has proportionally the right number of points. From this definition it follows that a sequence $\{x_n\}$ is uniformly distributed if for all Riemann integrable functions, f, defined on I^s it holds that

$$lim_{N\to\infty}\frac{1}{N}\sum_{n=1}^{N}f(x_n) = \int_{I^s} f(x)dx. \tag{2}$$

It follows from the Central Limit Theorem that a sequence of independent, identically distributed points chosen from the uniform distribution on the interval I^s is indeed a uniformly distributed sequence.

Practically, it is only possible to deal with a finite number of integration nodes, so it is necessary to define some measure of uniformity for finite point sets. Such a quantity is known as the discrepancy. For a set $J \subset I^s$ and a sequence of N points $\{x_n\}$ in I^s, define this measure of the deviation of the point set, $\{x_n\}$, from being uniform in the set $J \in I^s$

$$R_N(J) = \frac{1}{N}\sum_{n=1}^{N}\chi_J(x_n) - m(J). \tag{3}$$

Various kinds of discrepancy can be defined then by restricting J to a certain class of subsets and taking a norm of R_N over this class. If E is the set of all rectangular parallelepipeds of I^s, then the L_∞ and L_2 norms are defined as:

$$D_N = sup_{J\in E}\left|R_N(J)\right|, \tag{4}$$

and

$$T_N = \left[\int_{(\mathbf{x},\mathbf{y})\in I^{2s}, x_i<y_i}(R_N(J(x,y)))^2 dxdy\right]^{\frac{1}{2}}. \tag{5}$$

Here $J(\mathbf{x},\mathbf{y})$ indicates the rectangular parallelepiped with opposite corners at $(\mathbf{x},\mathbf{y})$. If E^* is the set of subrectangles with one corner at $\mathbf{x} = \mathbf{0}$, then the traditional measures, the star discrepancies are defined as:

$$D_N^* = sup_{J\in E^*}\left|R_N(J)\right|, \tag{6}$$

and

$$T_N^* = \left[\int_{I^s}(R_N(J(x)))^2 dx\right]^{\frac{1}{2}}. \tag{7}$$

Here $J(x)$ is the rectangular parallelepiped with one corner at $\mathbf{0}$ and the opposite corner at x.

Equation 5 is only a theoretical definition of the L_2 discrepancy, [19] gave a practical equation for calculating this L_2 discrepancy.

$$T_N^2 = \frac{1}{N^2}\sum_{n=1}^{N}\sum_{m=1}^{N}\prod_{i=1}^{s}(1-max(a_{n,i},a_{m,i}))\times min(a_{n,i},a_{m,i}) - \frac{2^{1-s}}{N}\sum_{n=1}^{N}\prod_{i=1}^{s}a_{n,j}(1-a_{n,i}) + 12^{-s} \tag{8}$$

The Halton [10], Soboĺ and Faure [8] sequences are well known and widely applied low-discrepancy sequences. They are very uniform in I^s, however they have some very poor two-dimensional projections. Fig. 1 shows a 2-D projection of Soboĺ sequence. The two dimensions are 27th and 28th. Clearly banded structures and large gap exists in the figure. The organized structures in Fig. 1 indicates that these two dimensions of the Soboĺ sequence are not very uniformly distributed. The organized shape and large gaps in Fig. 1 may cause biased samples in quasi-Monte Carlo applications. Scrambling is a solution for these poor low-dimensional projections.

2.2 Generating the Soboĺ Sequence

Before discussing how to scramble the Soboĺ sequence, let us first look at how an unscrambled Soboĺ sequence is created so that we can identify the important characteristics of Soboĺ sequence that we will use in its scrambling. The Soboĺ sequence, as originally defined by Soboĺ, is generated from a set of special binary vectors of length w bits, v_i^j, $i = 1, 2, \cdots, w$, $j = 1, 2, \cdots, d$. These numbers, v_i^j, are called direction numbers. In order to generate direction numbers for dimension j, we start with a primitive

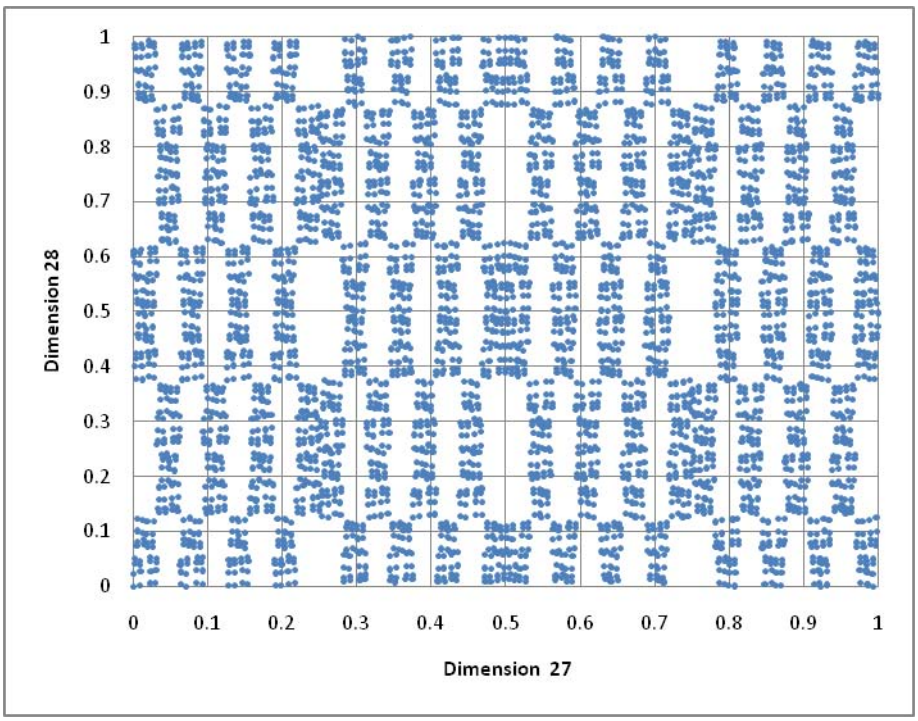

Figure 1: A Poor 2-D Projection of the Soboĺ Sequence

(irreducible) polynomial over the finite field $\mathcal{F}_2$ with elements $\{0, 1\}$. Let us suppose the primitive polynomial used in generating dimension j of the Soboĺ sequence is

$$p_j(x) = x^q + a_1 x^{q-1} + ... + a_{q-1}x + 1. \tag{9}$$

Once we have chosen this polynomial, we use its coefficients to define a recurrence for calculating v_i^j , the direction number in dimension j. It is generated using the following q-term recurrence relation:

$$v_i^j = a_1 v_{i-1}^j \oplus a_2 v_{i-2}^j \oplus \cdots \oplus a_{q-1} v_{i-q+1}^j \oplus v_{i-q}^j \oplus (v_{i-q}^j / 2^q), \tag{10}$$

where $i > q$, $\oplus$ denotes the bitwise `XOR` operation, and the last term is v_{i-q} shifted right q places. The initial numbers $v_1^j \cdot 2^w$, $v_2^j \cdot 2^w$, $\cdots$,$v_q^j \cdot 2^w$ can be arbitrary odd integers smaller than 2, 2^2,..., and 2^q, respectively. The Soboĺ sequence x_n^j ($n = \sum_{i=0}^{w} b_i 2^i, b_i \in \{0, 1\}$) in dimension j is generated by

$$x_n^j = b_1 v_1^j \oplus b_2 v_2^j \oplus \cdots \oplus b_w v_w^j \tag{11}$$

We should use a different primitive polynomial to generate Soboĺ sequence in each dimension.

Generating the Soboĺ sequence as defined by equations **??** is tedious and time consuming. Antonov and Saleev [1] provided another efficient way to calculate the Soboĺ sequence. They proved that taking

$$x_n^j = g_1 v_1^j \oplus g_2 v_2^j \oplus \cdots \tag{12}$$

where $\cdots g_3 g_2 g_1$ is the Gray code representation of n does not affect the asymptotic discrepancy. For $k = 2, 3, \cdots$, this shuffles (permutes) each initial segment of length 2^k of Soboĺ's original sequence. We remind the reader of the following properties of the Gray code:

- The Gray code for n is obtained from the binary representation of n using $\cdots g_3 g_2 g_1 = \cdots b_3 b_2 b_1 \oplus \cdots b_4 b_3 b_2$.

- The Gray code for n and the Gray code for $n+1$ differ in only one bit. If b_c is the rightmost zero-bit in the binary representation of n (add a leading zero to n if there are no others), the g_c is the bit whose value changes.

Using these properties, and defining x_n^j by equation 12, we can calculate x_{n+1}^j in terms of x_n^j as

$$x_{n+1}^j = x_n^j \oplus v_c^j, \tag{13}$$

where b_c is the rightmost zero-bit in the binary representation of n. The Antonov-Saleev method is thus faster than Soboĺ's original scheme, and so we will use Antonov-Saleev method to implement the original Soboĺ sequence.

3 Scrambling the Soboĺ Sequence

This section discusses about our scrambling algorithm and its implementation in `SPRNG`. First we briefly introduce some currently used scrambling algorithms.

3.1 Current Scrambling Methods

Owen investigated a general approach to scrambling of nets and sequences in [23]. His scrambling scheme can be described as follows. Let $b \geq 2$ be an integer. Let δ is a mapping from interval $[0,1)$ to the b-ary representation of $\delta(A) \in [0,1)$ determined in the following way. Let $A = a_1 b^{-1} + a_2 b^{-2} + \cdots$, where $a_1, a_2, \cdots$ are in $\{0, 1, \cdots, b-1\}$. Next, for each possible value of a_1, we fix a permutation π_{a_1} of $\{0, 1, ..., b-1\}$, and define the second b-ary digit as $\pi_{a_1}(a_2)$. We can continue in this way with the definition of the third digit, fourth digit, and so on, and so obtain $\pi_{a_1,a_2}(a_3)$, $\pi_{a_1,a_2,a_3}(a_4)$, $\cdots$. In Owen's scrambling scheme, each permutation is uniformly distributed over the $b!$ possible permutations, and the permutations are mutually independent. In s dimensions, we consider an s-tuple of b-ary scramblings $(\delta_1, \cdots, \delta_s)$. Owen showed that if $\delta_1, \cdots, \delta_s$ are chosen as fully random and mutually independent. then the s-dimensional scramblings preserve the properties of (t, m, s)-nets and (t, s)-sequences. A consequence of this is a sequence and its scrambling under this scheme have the same asymptotic discrepancy.

Based on Owen's scrambling scheme, many scrambling other algorithms [22, 14, 2, 11] that randomize a single digit at a time have been proposed. Alternatively, [6] proposed a multiple digital scrambling approach using a popular pseudorandom number generator as a scrambler, namely the **L**inear **C**ongruential **G**enerator (LCG). From now on we call this algorithm linear scrambling. The procedure is described as follows:

- $y_n = \lfloor x_n \times 2^k \rfloor$, is the k most-significant bits of C, the number to be scrambled.
- $y^* = a y_n (mod\ m)$ and $m \geq 2^k - 1$, is the linear scrambling applied to this integer.
- $z_n = \frac{y_n^*}{2^k} + (x_n - \frac{y_n}{2^k})$, is the final scrambled form derived from x_n.

The LCG is the key to the success of this scrambling method. LCGs with both power-of-two and prime moduli are common pseudorandom number generators. When the modulus of an LCG is a power-of-two, the implementation is cheap and fast due to the fact that modular addition and multiplication are just ordinary computer arithmetic when the modulus corresponds to the size of a computer word. The disadvantage, is in terms of quality, as it is hard to obtain the desired quality of pseudorandom numbers when using a power-of-two as modulus LCG. The linear scramblings thus used prime moduli, but only primes with special form such as the Mersenne or a Sophie-Germain primes.

3.2 Scrambling via Permutation

Owen's scrambling scheme randomizes each b-ary number a_n by uniformly choosing permutations $\pi_{a_1,a_2,\cdots,a_{n-1}}$ of $a_1, a_2, \cdots, a_{n-1}$. And linear scrambling produce randomization through multiplication with a pseudorandom integer. Our new algorithm can be regarded as a variant of Owen's scrambling scheme. It uses a random number to choose a permutation randomly and uniformly, then the permutation is applied to a certain number of digits for scrambling. The algorithm is described in detail as the follows.

Let Ω_w denote the set of all permutation of the numbers $\{0, 1, \cdots, w-1\}$, obviously the cardinality of Ω_w is $w!$. A bijection, ϕ^w ,is defined as $\phi^w : \{0, 1, \cdots, w!-1\} \to \Omega_w$, so that for any $a \in \{0, 1, \cdots, w!-1\}$, $\phi^w(a)$ is a permutation of the numbers $\{0, 1, \cdots, w-1\}$. To simplify notation, we use ϕ_a^w to denote $\phi^w(a)$. Assume that x_n^j is the Soboĺ number in the jth dimension corresponding to n as used in § 2.2. Then $x_n^j = g_0 \times (2^k)^0 + g_1 \times (2^k)^1 + g_2 \times (2^k)^2 +, \cdots$, where $k \geq 1$. If $r_{n,j}$ is the nth random number from random number sequence, j, and $r_{n,j} \in \{0, 1, 2, \cdots, w! - 1\}$, then the scrambled Soboĺ number is

$$x_n^{j*} = \phi_{r_{n,j}}^{2^k}(g_0) \times (2^k)^0 + \phi_{r_{n,j}}^{2^k}(g_1) \times (2^k)^1 + \phi_{r_{n,j}}^{2^k}(g_2) \times (2^k)^2 +, \cdots . \tag{14}$$

The idea behind this algorithm is to replace each k-bit digit in x_n^j by the permutation $\phi_{r_{n,j}}^{2^k}$. We represent the Soboĺ number in binary because Soboĺ sequence generation is done with operations in the finite field $\mathcal{F}_2$. The algorithm can be easily modified to handle other low-discrepancy sequences represented in bases other than 2, e.g. the prime number based Halton sequence.

One problem in this algorithm is that for each iteration we need to permute multiple groups of digits to scramble all the digits. Since the first k significant bits play a key role in deciding the value of the number, we can get by with only significant the most significant k bits, i.e. given x_n^j, there is an integer $m \geq 0$ which satisfies $x_n^j = g_0 \times (2^k)^0 + g_1 \times (2^k)^1 + g_2 \times (2^k)^2 +, \cdots, +g_m \times (2^k)^m$ and $g_m \neq 0$. The scrambled Soboĺ number is thus

$$x_n^{j*} = g_0 \times (2^k)^0 + g_1 \times (2^k)^1 +, ..., +g_{m-1} \times (2^k)^{m-1} + \phi_{r_{n,j}}^{2^k}(g_m) \times (2^k)^m. \tag{15}$$

The algorithm can be further extended to replace the first p significant k-bit digits through permutations, $m \geq p \geq 1$. Algorithm 1 illustrates our Soboĺ scrambling algorithm.

1. Generate the nth Soboĺ number in dimension j: x_n^j;
2. Disassemble x_n^j into $\{g_0, g_1, \cdots, g_m\}$;
3. Get a random number $r_{n,j}$ from a pseudorandom number generator;
4. Choose a permutation $\phi_{r_{n,j}}$ from the permutation set Ω_{2^k};
5. Replace $\{g_0, g_1, \cdots, g_m\}$ with $\{g_0, \cdots, g_{m-p}, \phi_{r_{n,j}}(g_{m-p+1}), \cdots, \phi_{r_{n,j}}(g_m)\}$;
6. Reassemble the Soboĺ number as
$x_n^j \leftarrow g_0 \times (2^k)^0 +, \cdots, +g_{m-p} \times (2^k)^{m-p} + \phi_{r_{n,j}}(g_{m-p+1}) \times (2^k)^{m-p+1} +, \cdots, +\phi_{r_{n,j}}(g_m) \times (2^k)^m$;

Algorithm 1: Our Algorithm for Scrambling the Soboĺ Sequence

3.3 Implementing the Scrambling in SPRNG

The algorithm 1 described in the previous section was implemented using the software infrastructure provide by the SPRNGlibrary. The SPRNGlibrary was designed to support parallel Monte Carlo applications on scalable and distributed architectures. It contains most of the important pseudorandom number generators, and many tools for statistical testing of pseudorandom number generators. Fig. 2 illustrates the software structure of SPRNGusing UML, the **U**nified M**M**odeling **l**anguage. The directory SRC is the major directory, which contains all the core implementations of the SPRNGlibrary. Codes

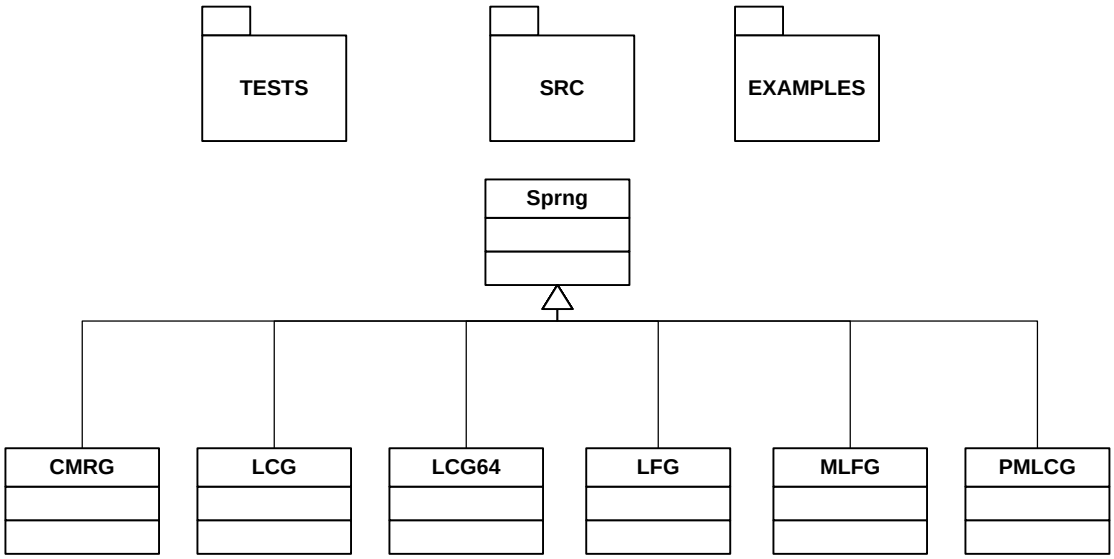

Figure 2: The Software Structure of SPRNG

for the different pseudorandom number generators are within the subdirectory SRC. SRC also includes some important utility tools like a prime number generator. Directory TEST is the directory containing different kinds of statistical tests for random number generators, including the collision test, coupon collector's test, equidistribution test. Directory EXAMPLES includes not only applications for demonstrating the usage of the SPRNGlibrary, but also a few driver programs for testing and debugging purposes.

The key class in SPRNGis the Sprng class, which is the parent class of all the other pseudorandom number generator classes. In other words, as long as the pseudorandom number generator class inherits and implements the Sprng class's interface, it can be merged into the SPRNGlibrary seamlessly. For this reason, our Soboĺ sequence generator class Soboĺ inherits the class Sprng, as demonstrated in Fig. 3. Considering that the low-discrepancy sequences are widely used in high dimensional integration, we extended the interface by adding a new class function get_rn_vector. The vector value returned by this function is a point in I^s.

Scrambler is a virtual class which defines the general interface for all the scrambler classes. LinearScrambler is an implementation of the scrambling algorithm in [6]. PermutationScrambler implements the algorithm 1.

PermuationFactory is a virtual class to generate the permutation used in step 4 of algorithm 1. Two child classes of PermuationFactory are implemented to provide real permutations for an input random number $r_{n,j}$. StaticFactory will generate and store all the $n!$ permutations for $\{0, 1, \cdots, n-1\}$ at initialization. For any integer $n! - 1 \geq r \geq 0$, StaticFactory retrieves the rth permutation and returns it.

Except for the cost of initializing all of the $n!$ permutations, StaticFactory is quite efficient, but clearly the n used must be confined to a small values. For the Soboĺ sequence, $2^2!$ and $2^3!$ are already big enough for scrambling the sequences as shown by the empirical evidence in § 4.

Unlike StaticeFactory, DynamicFactory generates one permutation according to the value of the random number $r_{n,j}$ at the time the class's function is called, not when the class is initialized. Theoretically it can generate any permutation of arbitrarily large Ω_w. In [24], a few fast algorithms to randomly generate permutations were introduced. DynamicFactory uses the algorithm published by R. Durstenfeld [7] to randomly generate permutations.

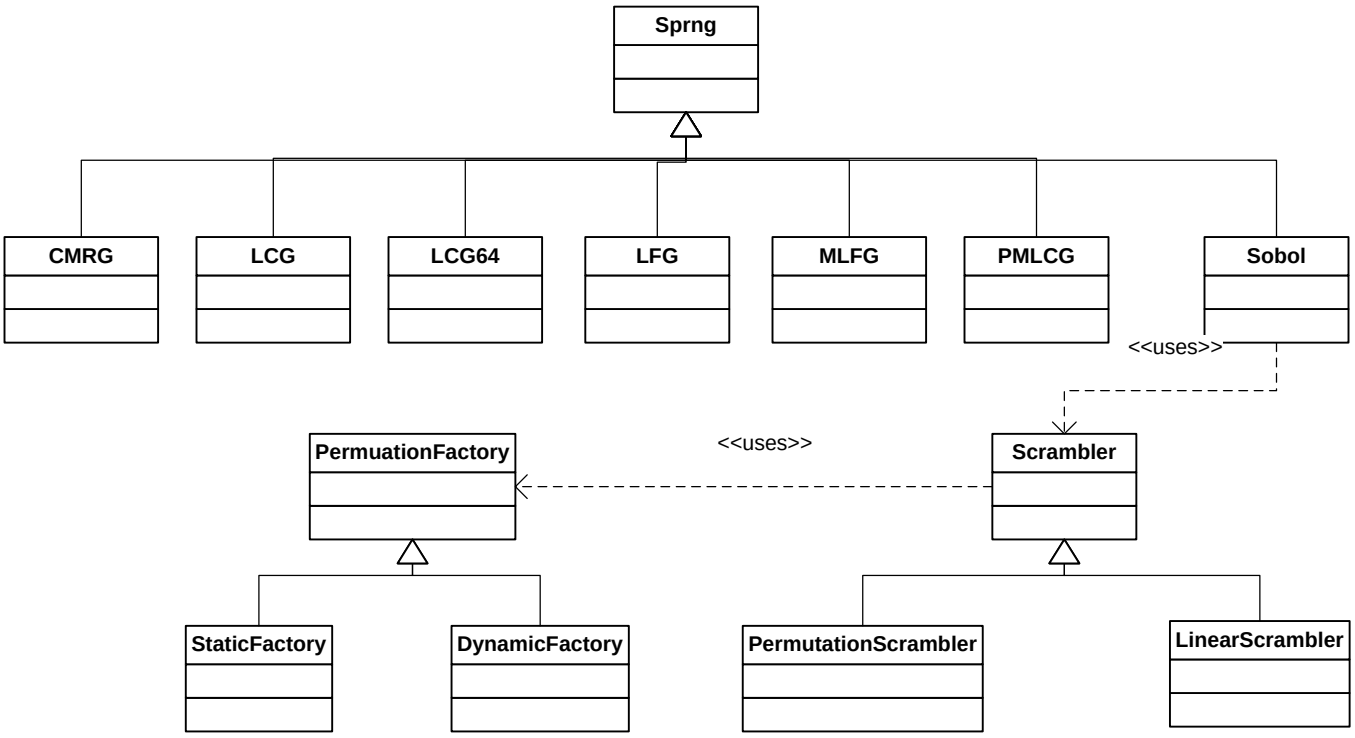

Figure 3: Soboĺ in SPRNG

CPU	*Memory*	*Compiler*	*OS*	SPRNG
Pentium(R) D 3.00 GHz	2GB	gcc 4.1.1	Linux 2.6.18-8.e15	version 4.0

Table 1: Experiment Configurations

4 Computational Experiments

We designed a group of experiments to verify the performance of our scrambling algorithm. The tests were executed on a Linux PC. The software and hardware configurations of the experiments are listed in table 1.

In the experiments, we used the LCG of SPRNGto generate pseudorandom numbers for picking a permutation from Ω_w. The experiments comprise two types: 2-D projections tests (to visually verify scrambled points) and the computation of the L_2 discrepancy test, where 5 is used to calculate it. Two parameters are critical in our scrambling algorithm, one is the number of bits to be replaced at a time (k in algorithm 1), and the other is the number of k-bit digits to be replaced in one iteration (p in Alg. 1). We use the notation $k \times p$ to denote these two parameters. For example, 2×8 means eight 2-bit digits (from most-significant bits to the least) will be replaced for each Soboĺ number.

4.1 2-D Projections

Fig. 1 demonstrates the banded structure and large gaps in the 2-D projection of 27th and 28th dimensions of the Soboĺ sequence. This illustrates the correlated relationship between those dimensions. Our scrambled Soboĺ sequence successfully breaks such such structures, as demonstrated in Fig. 4 and Fig. 5. In Fig. 1, 4096 points were generated, and the 27th and 28th dimensions were projected on the plane in Fig. 4 and Fig. 5. In the left figure of Fig. 4, although most of the banded structures have been destroyed, a few still exist in the center, left-bottom and right-top corner. The reason for this is that

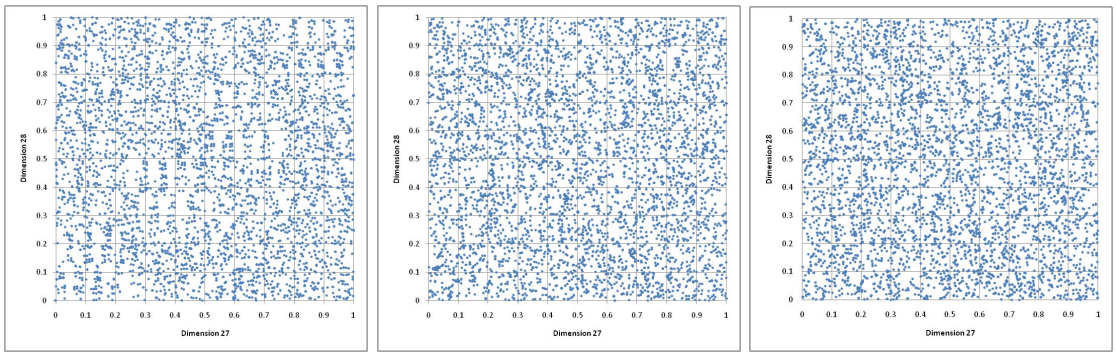

Figure 4: A 2-D Projection of Scrambled Soboĺ Sequence. Left: 2×1; Middle: 2×8; Right: 2×15

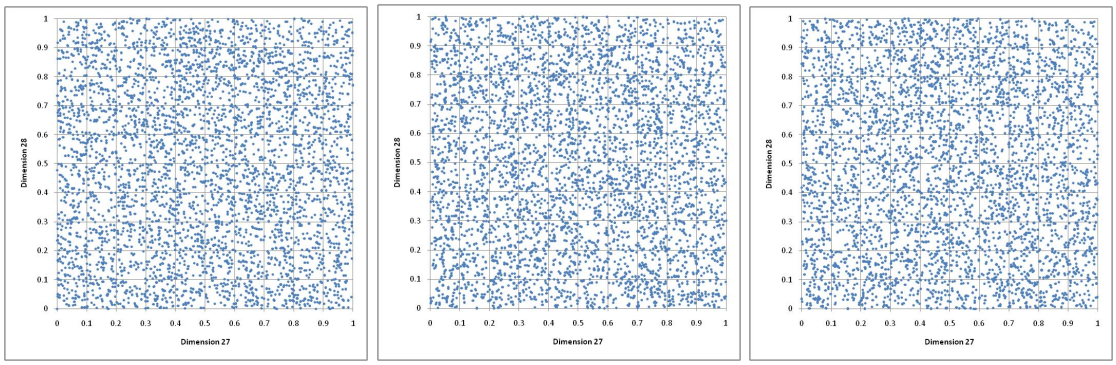

Figure 5: A 2-D Projection of Scrambled Soboĺ Sequence. Left: 3×1; Middle: 3×5; Right: 3×10

only randomizing the 2 most-significant bits could not effectively scramble the Soboĺ sequence. This is verified by the middle and right graphs of Fig. 4. In the middle graph of Fig. 4, the banded structures have totally disappeared after half of bits were scrambled. But it is enough to randomize three bits at one time to destroy the correlations between the 27th and 28th dimensions as demonstrated in left graph of Fig. 5.

4.2 The L_2 Discrepancy

This section explores the empirical relationship between number of points and L_2 discrepancy in terms of our scrambling algorithm. We plot the logarithm of both of number of points and of L_2 discrepancy, so that the two values produce a linear relation in the graph if the L_2 discrepancy obeys a power-law relationship with the number of points. In that case, the slope in the log-log plot is the exponent. All of the tests were done on two Soboĺ sequences: one is 10-dimensional and the other is 100-dimensional. Fig. 6 illustrates the linear relation between L_2 discrepancy and number of points for the 10- and 100-dimensional Soboĺ sequences. We scrambled the Soboĺ sequences with different parameter configurations, and for each parameter configuration, we generated four scrambled streams. There were 16384 points created for each scrambled or unscrambled Soboĺ sequence. The interval $[1, 16384]$ is equally divided into 32 subintervals, and the L_2 discrepancy is calculated on those boundary points of the 32 subintervals that are $512, 1024, \cdots, 16384$.

Fig. 7 and Fig. 8 are for the scrambled 10-dimensional Soboĺ sequences with 8 groups and 16 groups of 2-bit permutations. Compared with the left graph of Fig. 6 (the unscrambled 10-dimensional Soboĺ sequence), all the eight graphs have a quite similar linear relationship. The slopes of the log-log plots

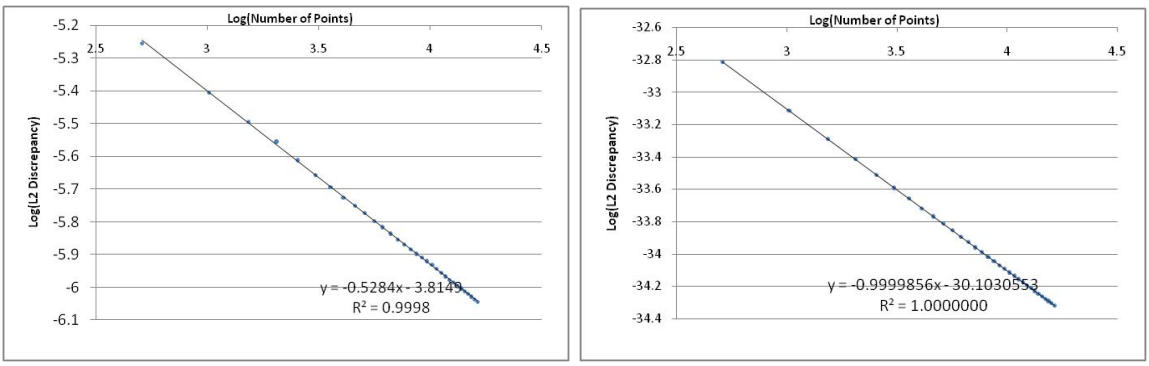

Figure 6: L_2 Discrepancy of Sobol' Sequences. Left: 10-Dimension Sobol'; Right: 100-Dimension Sobol'

Dimensions	*Unscrambled*	2×8	2×15	3×5	3×10
10	0.005999	0.024497	0.029746	0.026246	0.031496
100	0.055993	0.302454	0.351697	0.314202	0.350697

Table 2: CPU Time for Generating 16384 Scrambled and Unscrambled Sobol' Points

are close to -0.5.

Similarly, Fig. 9 and Fig. 10 are for scrambled 10-dimensional Sobol' sequences with 5 and 10 groups of 3-bit permutations. The slopes of the log-log plots are also close to -0.5, but a little bigger than -0.5. And the result of 5 groups scrambling is very similar to 10 groups scrambling.

The next four figures are all for scrambled 100-dimensional Sobol' sequences. Fig. 11 and Fig. 12 are for 8 groups and 15 groups of 2-bit permutations. Fig. 13 and Fig 14 are for 5 groups and 10 groups 3-bit permutations. Compared with right graph of Fig. 6 (the unscrambled 100-dimensional Sobol' sequence), all the 8 streams of 2-bit scramblings (Fig. 11 and Fig. 12) have a very similar linear relationship to that of original Sobol' sequence – the slope values are close to -1. But for 3-bit scramblings (Fig. 13 and Fig. 14), only two of eight streams have a linear relationship similar to that of the unscrambled Sobol' sequence (right graph of Fig. 6). To further investigate this, we put all of the L_2 discrepancy data of 3-bit scramblings in Fig 14 together with that of the unscrambled Sobol' sequence into Fig. 15, and we used the number of points, not the logarithm, on the x-axis of Fig. 15. All the scrambled sequences have lower discrepancy than the unscrambled sequence does. The scrambled sequence (**stream 3** in Fig. 15 and Fig. 14) whose linear relation is similar to that of unscrambled sequence has a more smooth shape than other scrambled sequences do.

Table 2 lists the time used for generating those scrambled and unscrambled Sobol' sequences in units of seconds. Clearly our scrambling algorithm is about 4 to 6 times slower than the unscrambled Sobol' sequence. But time used for generating the scrambled 100-dimension Sobol' sequence is almost 10 times that of generating the scrambled 10-dimension Sobol' sequences. This means that our algorithm appears to scale linearly with dimension. We also want to mention that we did not optimize the implementation of our algorithm.

5 Conclusions

We provide an algorithm to effectively randomize the Sobol' sequence vi multi-bit permutation. The advantage of this algorithm is that randomization and scrambling speed can be balanced through setting different parameter configurations – k and p. Experimental results verify that our algorithm not only

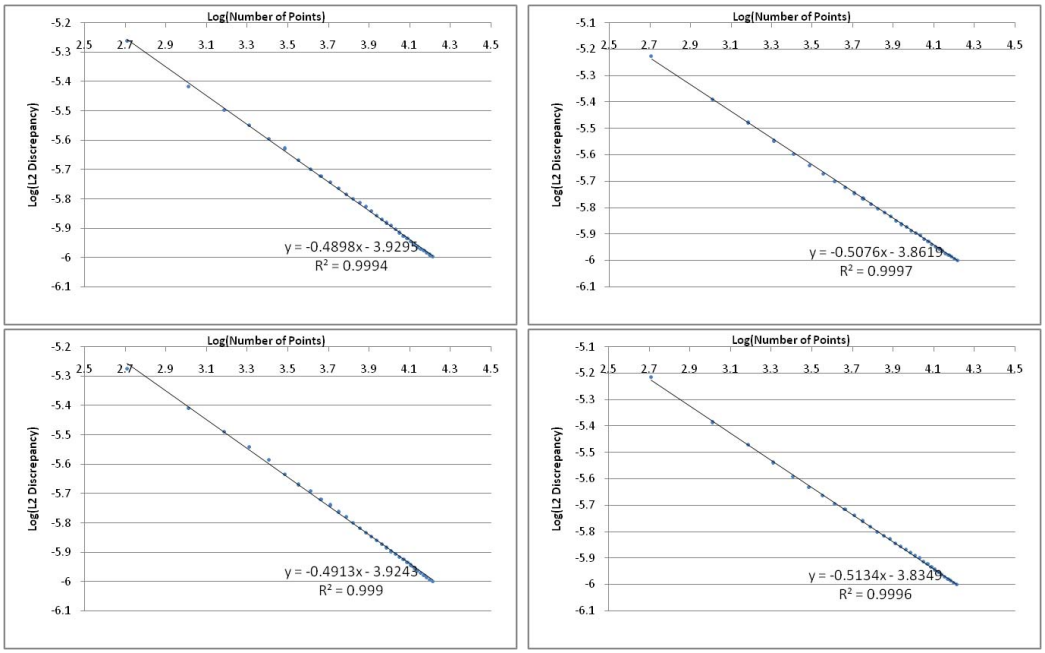

Figure 7: L_2 Discrepancy of Scrambled 10-Dimension Soboĺ 2×8

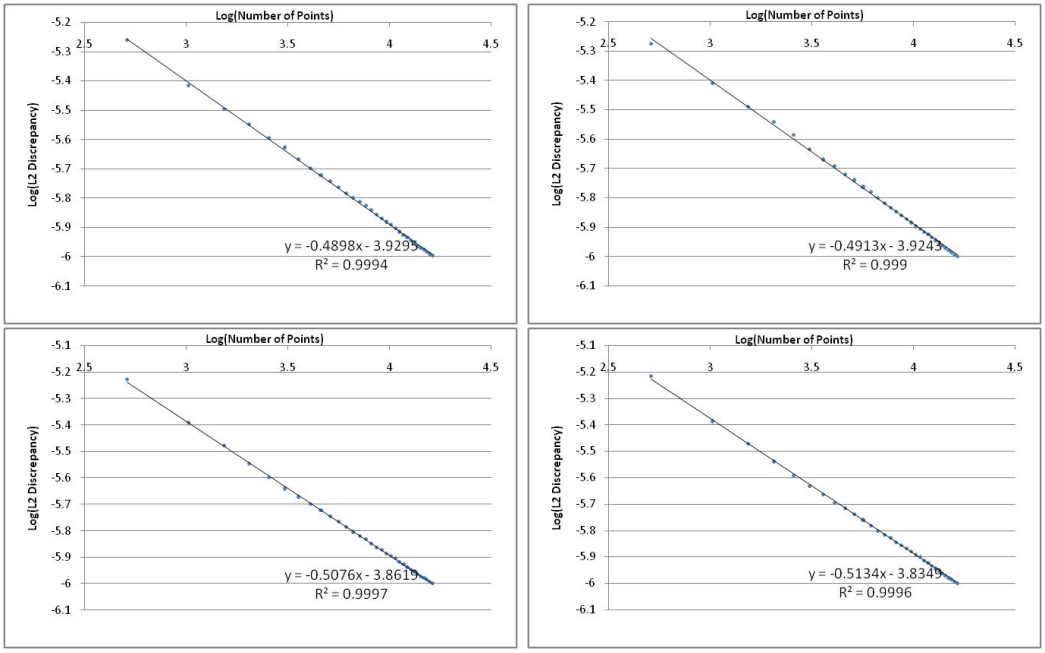

Figure 8: L_2 Discrepancy of the Scrambled 10-Dimensional Soboĺ 2×15

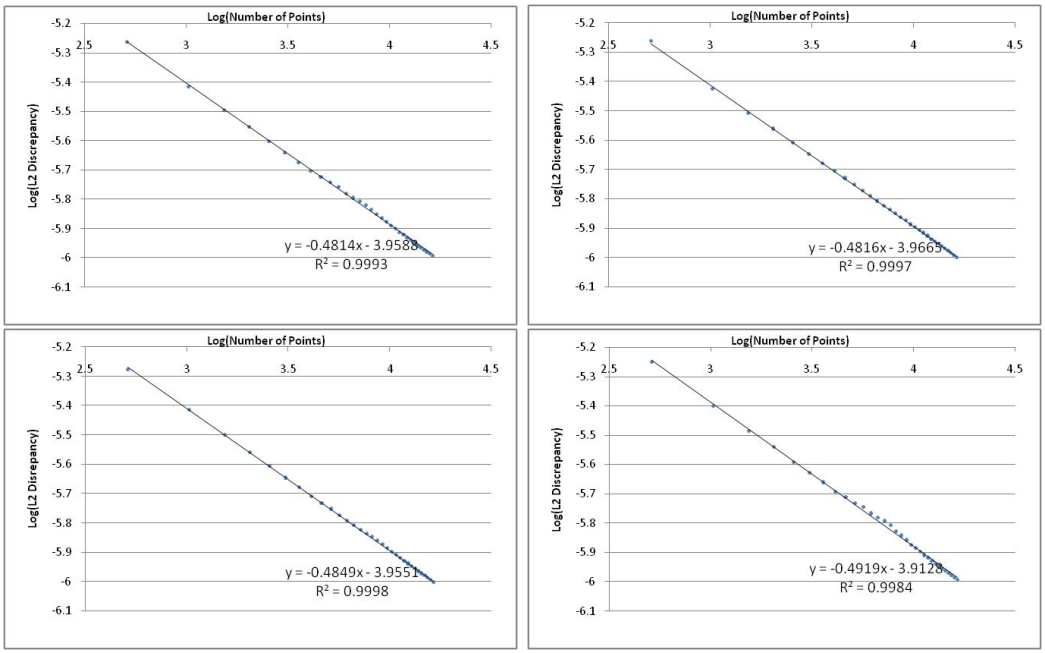

Figure 9: L_2 Discrepancy of the Scrambled 10-Dimensional Soboĺ 3×5

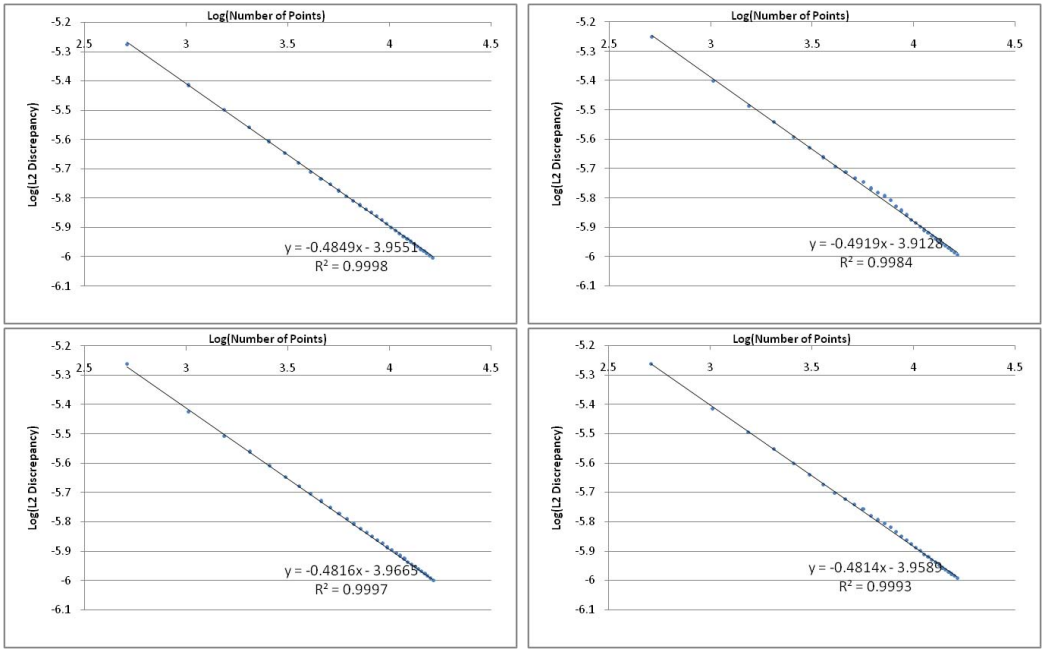

Figure 10: L_2 Discrepancy of the Scrambled 10-Dimensional Soboĺ 3×10

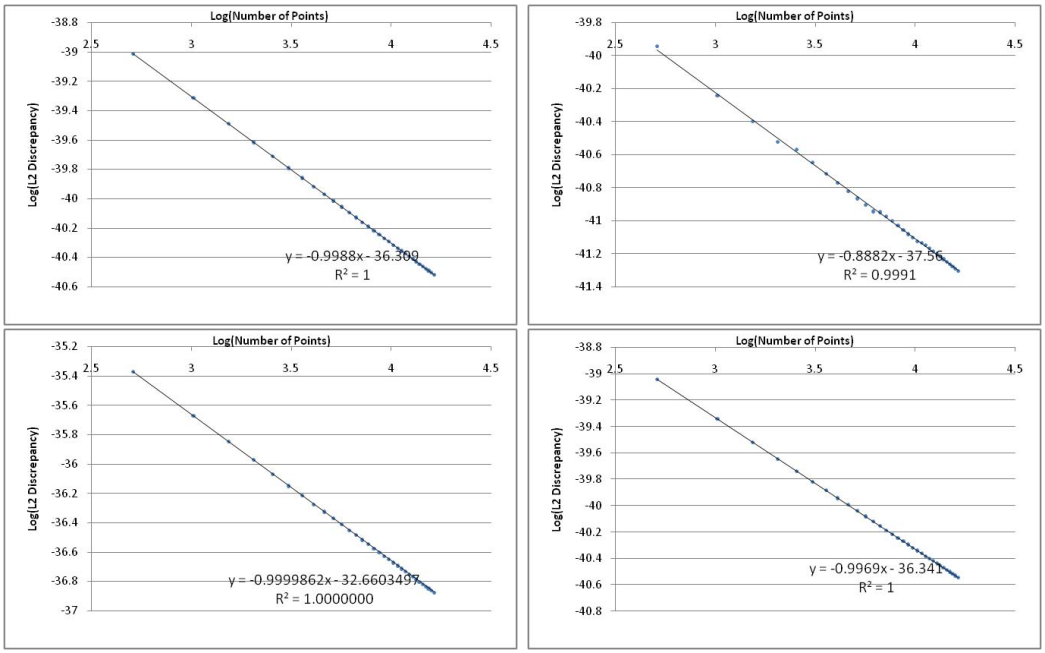

Figure 11: L_2 Discrepancy of the Scrambled 100-Dimensional Soboĺ 2×8

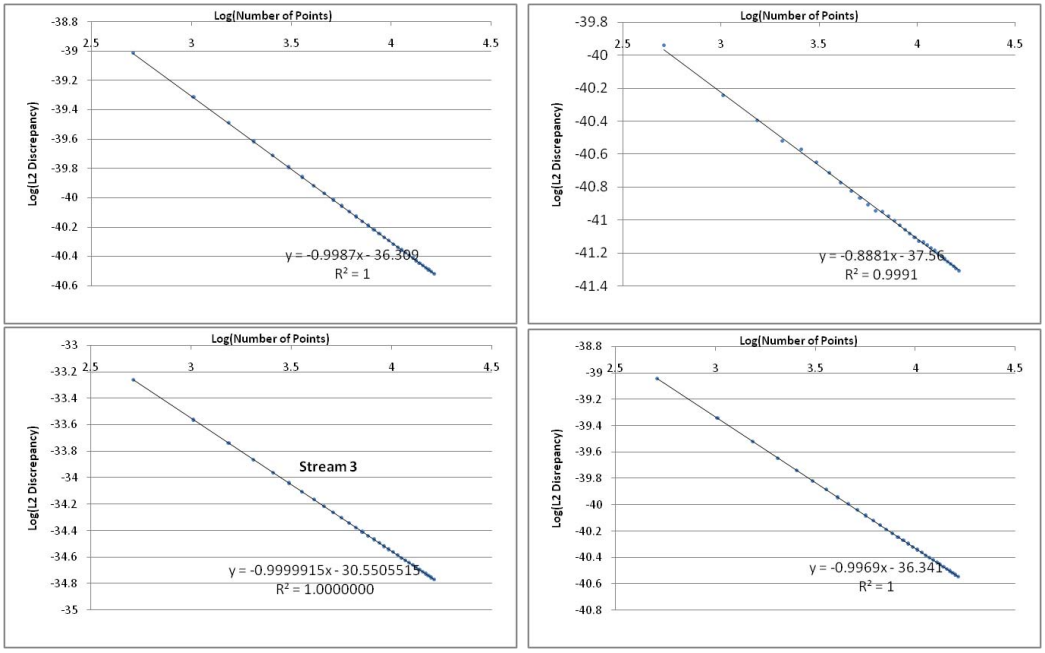

Figure 12: L_2 Discrepancy of the Scrambled 100-Dimensional Soboĺ 2×15

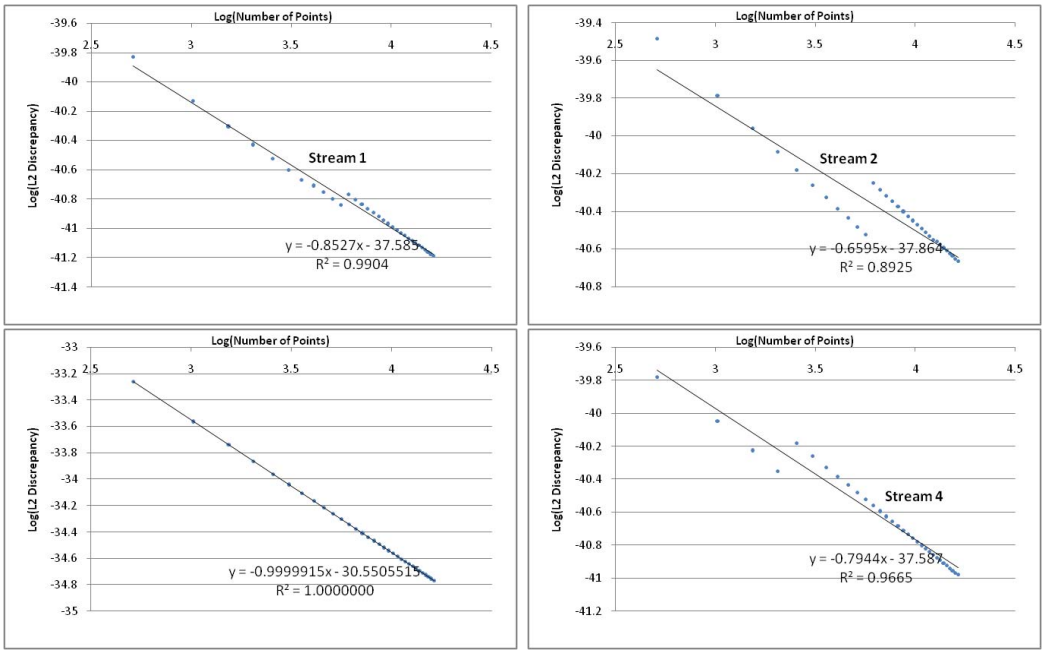

Figure 13: L_2 Discrepancy of the Scrambled 100-Dimensional Soboĺ 3×5

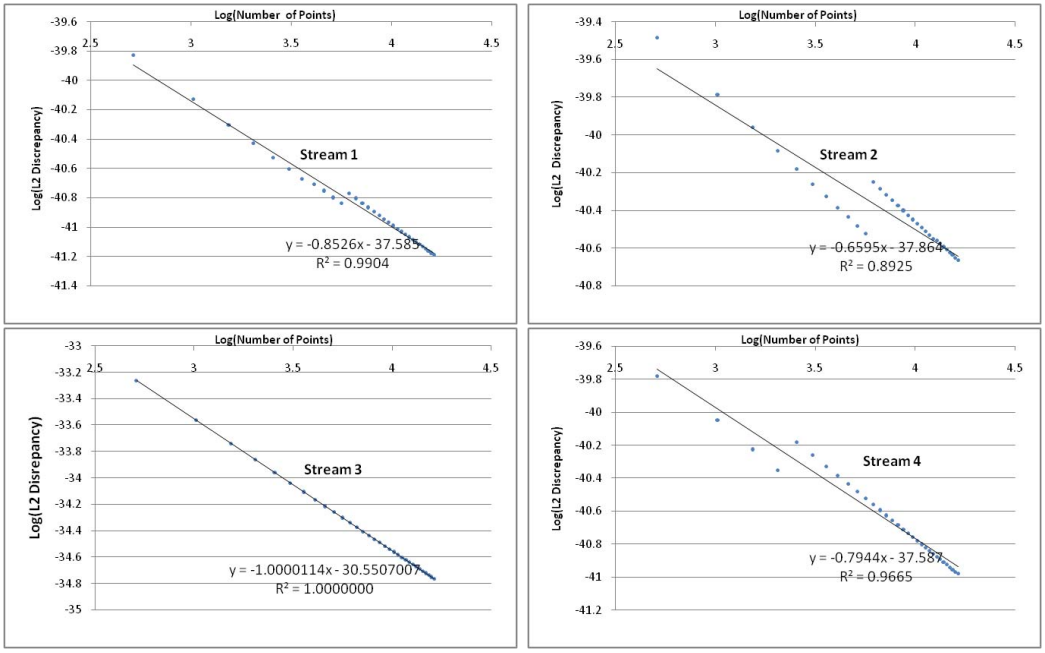

Figure 14: L_2 Discrepancy of the Scrambled 100-Dimensional Soboĺ 3×10

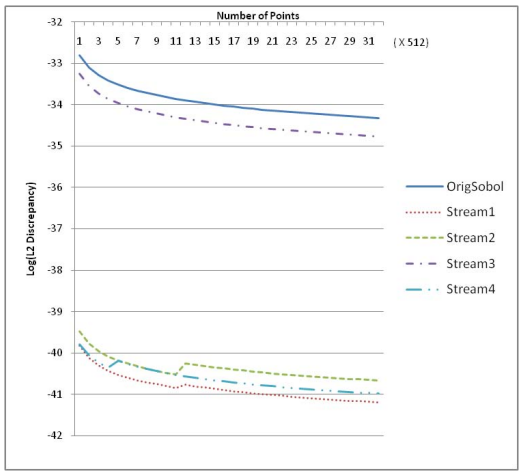

Figure 15: L_2 Discrepancy of the Unscrambled and All the Scrambled (3×10) 100-Dimensional Soboĺ Sequences

effectively breaks the correlations between dimensions, but also has a low discrepancy similar to the unscrambled Soboĺ sequence. While our method is not strictly an Owen scrambling, this confirms that it behaves as such.

With certain modifications, the scrambling algorithm can also be applied to other low-discrepancy sequences such as the Halton and Faure sequences. One problem for applying the algorithm to other quasirandom sequences such as the Halton sequence, is how to generate randomly and efficiently the large number permutations required for large bases. It is also important to point out that our current implementation has not been optimized. We plan to do this in the future. But we also need to develop a measure to evaluate how the algorithm's randomization process effects the discrepancy properties of the sequences, or in other words, how to derandomize the Soboĺ sequences scrambled by our algorithm.

References

[1] I. A. Antonov and V. M. Saleev. An economic method of computing l_{p_τ}-sequences. *USSR Comput. Math. and Phy.*, 19:252–256, 1979.

[2] Emanouil I. Atanassov. A new efficient algorithm for generating the scrambled sobol' sequence. In *NMA '02: Revised Papers from the 5th International Conference on Numerical Methods and Applications*, pages 83–90, London, UK, 2003. Springer-Verlag.

[3] R. E. Caflisch. Monte Carlo and Quasi-Monte Carlo Methods. *Acta Numerica*, **7**:1–49, 1998.

[4] H. Chi and A. Lorenzo. A Parallel Cellular Automata Model to Forecast the Effects of Urban Growth in North Florida. *The Sixth IMACS Seminar on Monte Carlo Methods*, Reading, UK, 2007.

[5] H. Chi and M. Mascagni. Efficient Generation of Parallel Quasirandom Sequences via Scrambling. *ICCS 2007/Lecture Notes in Computer Science*, Beijing, China, 2007.

[6] Hongmei Chi, Peter Beerli, Deidre W.Evan, and Micheal Mascagni. On the scrambled sobol sequence. In *ICCS2005*, pages 775–782, 2005.

[7] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964.

[8] H. Faure. discrepance de suites associees a un systeme de numeration (en dimension s). *Acta Arithmetica*, 41:337–351, 1982.

[9] B. Vandewoestyne H. Chi, M. Mascagni and R. Cools. An Empirical Investigation of Different Scrambling Methods for Faure Sequences. *The Sixth IMACS Seminar on Monte Carlo Methods*, Reading, UK, 2007.

[10] J. H. Halton. on the efficiency of certain quasirandom sequences of points in evaluating multidimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[11] Hee Sun Hong and Fred J. Hickernell. Algorithm 823: Implementing scrambled digital sequences. *ACM Trans. Math. Softw.*, 29(2):95–109, 2003.

[12] I.M.Sobol. On the distribution of points in a cube approximate evaluation of integrals. *USSR Comput. Math. and Phy.*, **7**(4):86–112, 1967.

[13] I.M.Sobol. Uniformly distributed sequences with additional uniformity properties. *USSR Comput. Math. and Phy.*, **16**:236–242, 1976.

[14] Jir?Matousek. On the l2-discrepancy for anchored boxes. *Journal of Complexity*, 14(4):527–556, 1998.

[15] Y. Li and M. Mascagni. Analysis of Large-scale Grid-based Monte Carlo Applications. *International Journal of High Performance Computing Applications*, **17(4)**:369–382, 2003.

[16] Y. Li and M. Mascagni. Grid-based Quasi-Monte Carlo Applications. *Monte Carlo Methods and Applications*, **11**:39–55, 2005.

[17] M. Mascagni. Scalable Parallel Random Number Gernerators Library (SPRNG) homepage:. `http://sprng.fsu.edu`, 1999-2007.

[18] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Transactions on Mathematical Software*, **26**:436–461, 2000.

[19] William J. Morokoff and Russel E. Caflisch. Quasi-random sequences and their discrepancies. *SIAM J. Sci. Comput*, 15:1251–1279, 1994.

[20] Harald Niederreiter. quasi-monte carlo methods and pseudo-random numbers. *Bull. Amer. Math. Soc.*, 6:957–1041, 1978.

[21] A. B. Owen. Randomly permuted (t, m, s)-nets and (t, s)-sequences. *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, 106 in Lecture Notes in Statistics:299–317, 1995.

[22] A. B. Owen. Randomly permuted (t,m,s)-nets and (t,s)-sequences. In *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pages 299–317, 1995.

[23] A. B. Owen. Monte carlo, quasi-monte carlo, and randomized quasi-monte carlo. In *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 86–97. Springer, 2000.

[24] Robert Sedgewick. Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164, 1977.

[25] S. Tezuka. *Uniform Random Numbers, Theory and Practice*. Kluwer Academic Publishers, Boston, 1995.